Genetic Programming of Music

Jeffrey B Putnam
New Mexico Insitute of Mining and Technology
Socorro, New Mexico
jefu@nmt.edu

August 30, 1994

Genetic Programming of Music

1 Introduction

Genetic Programming [2] is an evolutionary programming technique that evolves functions over time. Genetic Programming can be used to solve optimization problems and discovery problems. One early application of Genetic Programming is Sims' [4] evolution of images using user feedback to provide a kind of objective function. This project was an attempt to use similar techniques to evolve "music". This was not an attempt to prove anything interesting about musical theory or structure, instead it was a simple experiment in seeing if Genetic Programming could be used in other human directed evolutionary contexts.

While the goal of the project was to build programs that would evolve something that would sound pleasant and interesting enough to be considered musical (on at least some level) not all the noises generated are either pleasant or interesting. Thus, while the word "music" will be used, it doesnt mean that the noises are at all musical.

Two programs were built in this project. The first generates noises by directly constructing waveforms. The second constructs sequences of notes.

2 Mechanism

The Genetic Programming code used in this project was written in C and used the Boehm [1] garbage collector to simplify memory management and enable the use of shared structures without undue overhead.

Each individual in a population consists of one or more gp functions. For example, an individual might have a function for computing amplitude and another for computing frequency. Functions are crossed with other functions with the same purpose in other individuals at crossover time. There are two possibilities here, that two individuals are selected and all the (corresponding) functions are crossed over, or that for each function two individuals are selected and the functions crossed. Since arguments can be made for either method, a run time switch was provided that would enable either crossover method to be used.

The populations are typically small (6 by 6 is the most common) and are arranged in a two dimensional array. This was done both to simplify the user interface, and to allow for local crossover. Crossover may then be local (in which the individuals selected as parents of a new individual are picked from locations adjacent to the location of the new individual) or global in which the parents are selected from the entire population. Local crossovers seem to do better in many cases at preserving diversity but convergence is slower. The choice between local and global crossover is again a runtime switch.

Finally, both tournament and proportional selection are possible and the program may be toggled between them with a runtime switch.

To generate the music or noises, the program generated an amplitude for each sample over the total time for the noise. Since ".au" files were used, this meant 8000 samples per second, typically for 5 or 10 seconds. These samples were written to a file to be played by the host machine's "play" program.

The basic function set used consisted of the basic arithmetic operations, trigonometric functions sin, cos, some assorted other functions including $random, exp, log, max, min$ and constants drawn from the range $[-1, 1]$.

3 Noises

The first program constructed built waveforms. Part of the intent was to produce a program that could produce noises without a priori assumptions about notes and the structure of music. We also hoped that the output of the program might manifest interesting tonal qualities.

The program itself evolved over time as different problems with the methods used became apparent. In this section we will trace the evolution of the program and point out some of the problems encountered and our reasons for the specific changes made.

The first version merely generated amplitudes from a single GP generated function f with time its sole parameter $signal(i) = f(t)$. This version produced little more than irritating noise and evolved (if at all) very slowly.

In an attempt to improve the programs performance, several (incremental) changes were made to the program. First, the generation method was changed - instead of using a single function, the amplitude was built by combining a frequency and amplitude component - both functions genetically programmed and depending on time :

$$signal(i) = amp(t)sin(2\pi freq(t)) \tag{1}$$

In pseudocode format, the functions looked like:

```
for i from 1 to sampleLength
  t = i / samplesPerSecond
  signal[i] = amp(t) sin(2 * PI * freq(t))
end for
```

This was a significant improvement, but still produced very noisy samples. Examination of the functions used revealed that both the amplitude and the frequency functions would often produce numbers that were essentially off the scale. There were several ways to cope with such numbers. One was to simply limit them, by taking the maximum (minimum) of the number with some acceptable extremal value - but this both lost information and didnt work well - tending to produce intervals of silence or single tones. Another way to cope with these large numbers would be to map them to some acceptable range. This was accomplished by using a function :

$$nlmap(x, \alpha, \beta, \gamma, spread) = \frac{\alpha + \gamma(\alpha - \beta)/(\beta - \gamma)e^{xspread}}{1 + (\alpha - \beta)/(\beta - \gamma)e^{xspread}} \tag{2}$$

This function has a horizontal asymptote at α when x tends toward negative infinity, a horizontal asymptote at γ when x tends toward infinity and takes on the value β when $x = 0$. The *spread* parameter can be used to adjust the slope of the function at 0 and hence the speed at which the function approaches the asymptotes. (Figure 3).

The frequency and amplitude raw results were fed through this function in order to map essentially infinite intervals into finite ones - thus, frequency was mapped into the interval $[8, 8000]$ (Herz) with a center at 256 Hertz:

$$freq(i) = nl(rawfreq(i), 8, 256, 8000, 0.01)$$

Thus modified, the program produced rather noises that were less irritating and somewhat more interesting, but still not very exciting.

At this point another problem became the limiting factor in the experiments. It is relatively easy to watch a program draw pictures then rate the pictures. The images are static, always available for comparison and can be seen immediately. While the images may not be interesting or exciting, they

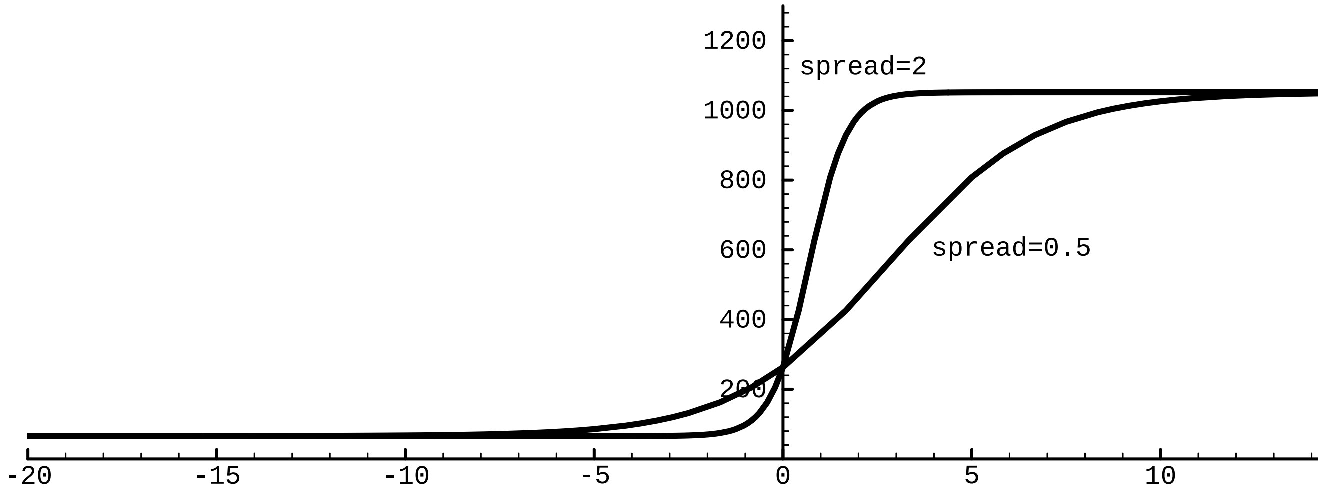

Figure 1: $nl(x, 0, 260, 1000, spread)$ for $spread = 0.5, 2.0$

are rarely irritating. However, with noises the situation is different. The noises must be listened to one at a time and cannot be compared as easily as images. They take a certain amount of time to play (and generate). Finally, many of the noises produced in early generations are very irregular, noisy and sometimes change loudness quite suddenly. In short, they are unpleasant and irritating and the process of listening to the noises and rating them is slow. This makes listener participation in the evolutionary process difficult.

These human factors mean that it is essential that the program produce at least minimally interesting noises early on in order to maintain the user's interest, and it helps to provide evolution that seems to be improving noticeably.

Given the human factors involved, it was crucial to make the program evolve interesting noises relatively quickly. Probably the best way to achieve this would be to run the program long enough to get interesting noises, then seed it at the start with a mix of these interesting noises and randomly generated individuals. However the program has not yet produced enough noises that are sufficiently interesting to make this feasible.

Another approach to this could involve running the program as a standard GP program trying to match sounds that are already available. This is a tempting experiment to try, but we have not yet attempted it, as we feel that the computational demands are probably beyond the local capacity.

Several changes were made to the program in attempts to make evolution move faster and to make the generated noises more palatable.

One was to add in some extra primitives that the functions could use. The most successful was to construct midpoint-displacement fractals with varying degrees of roughness. Several such were generated at program startup and a function `(fractal i t)` was constructed that selected sample number t out of the fractal in location i. This was tried because simple experiments showed that reasonably pleasant noises could be produced merely by using one fractal to produce a frequency and another to produce amplitude (without genetic programming).

However, this turned out to produce more noise - the reason being that the selection of which fractal is chosen can depend on a function (`(fractal (some-function) (...))`) and small changes in the value returned by `(some-function)` could lead to quite large changes in the output of the calling function which led to wildly discontinuous output. To avoid this problem, a variant of this was tried with a function for each sample (that is: `(fractal-`i `sample)` for several values of i) with much more success.

This may have been a case where thing worked too well, usually when the program was run with this function set it converged quickly to functions dominated by the fractals. This was good in that it produced things that you could listen to, but bad in that it was dependent on the fractal used - and the dominance of the fractals led to a slowing of evolution.

Another experiment tried to emulate the fractals, but with more control. In this version, each individual had both the amplitude and frequency functions and several other functions that computed preliminary samples - each function was iterated as in Equation 1 and the results stored. Each such function could use the results computed in any of the previous samples in its own computation. This was not a major improvement and led to much longer computation times.

Several other variants were tried, including ways to somehow smooth out the process of selecting a single sample amplitude from previously generated samples. None were successful enough to be kept. Computing FFTs and inverse FFT's was also tried, but this slowed the generation process down even more with minimal positive results.

4 Music

Faced with the problems of slow generation times, the irritating noises produced and relatively slow evolution, the direct generation of waveforms was (at least temporarily) abandoned in favor of an approach that would generate music in the form of sequences of notes. This had two main advantages over the waveform methods.

First, actually generating the notes was much simpler, the parameters for the note were computed by the genetic programming functions, then the note itself was generated. This usually meant thousands of times fewer evaluations of the genetic programming functions.

Second, since the notes were typically generated using sinusoidal waveforms they were not as grating as the random, noisy waveforms found in the previous experiments.

In this program each note is parameterized by an amplitude, a frequency, a duration, and a delay until the next note. These are all computed by gp functions and the results mapped into "reasonable" but essentially arbitrarily chosen intervals. The values of these parameters is saved in a variable for possible use in the computation of the next note. An additional three variables `x, y, z` are also provided, and are computed and saved in the same way. Pseudocode for the generation process is shown in Figure 2. Each of the functions `freq-nl, amp-nl` etc, are calls to `nlmap` Equation 2 with heuristicly chosen parameters. For example, frequency is mapped to the interval $[263/4, 263 * 4]$ with 0 mapped to 263 on the advice of a user.

The procedure generate note produces a waveform of the specified duration using Equation 1 but with the frequency and amplitude fixed. This means that each note is a pure sine wave with a fixed amplitude. While it was certainly feasible to add in parameters to specify attack and decay times, as well as a parameter to add in different modulations, we felt that the simpler model would work better for a first try.

In practice it turns out that this process should be run a number of times (100 seems to be a good choice) generating notes that are thrown away before actually saving the generated notes. This avoids startup transients which are often not typical of the longer term behavior of the system.

It was immediately clear that the note based system was much more interesting and usable than the waveform based system. The "music" generated was no longer distinctly grating, the generation times were short enough to

```
time = 0
oldX = oldY = oldZ = oldFreq = oldAmp = 0
oldDuration = oldDelay = 0
while time < maxTime do
  x = xfun(oldX, oldY, oldZ, oldFreq, oldAmp,
      oldDuration, oldDelay)
  y = yfun(oldX, oldY, oldZ, oldFreq, oldAmp,
      oldDuration, oldDelay)
  z = zfun(oldX, oldY, oldZ, oldFreq, oldAmp,
      oldDuration, oldDelay)
  freq = freqFun(oldX, oldY, oldZ, oldFreq, oldAmp,
      oldDuration, oldDelay)
  amp = ampFun(oldX, oldY, oldZ, oldFreq, oldAmp,
      oldDuration, oldDelay)
  duration = durationFun(oldX, oldY, oldZ, oldFreq, oldAmp,
      oldDuration, oldDelay)
  delay = delayFun(oldX, oldY, oldZ, oldFreq, oldAmp,
      oldDuration, oldDelay)
  generateNote(time, freq-nl(freq), amp-nl(amp), dur-nl(duration))
  time += dur-nl(duration) + dur-nl(delay)
  oldX = x
  oldY = y
  oldZ = z
  ....
end while
```

Figure 2: Pseudocode for Generating Notes

make it bearable over at least short periods of time, and there seemed (at least) to be evolution toward more interesting samples.

5 Interface

Both programs were written in C using essentially the same genetic programming base code. They generated ".au" files using the Sun "libaudio" library which could then be played with the "play" program. The programs could also dump out and read in the Lisp-like code for the functions generated, and so could be restarted easily.

The programs generated and wrote the sound files directly and the only interface needed for the user was textual. In this way the programs could be run on remote machines as well as local machines and could take advantage of faster computers that might not have sound capabilities to do the computation. This interface was not, however, particularly usable.

A TCL/TK [3] interface for X windows was built. This interface is constructed so that it communicates with the back end using text - so the back end did not need to be altered in any significant way. There is a cell corresponding to each individual in the population - since it may take some time to generate an individual for the user to play, the cells are marked when the sample for that individual cell is ready. This interface allows the user to play specific individuals (by running the "play" program) and rate them using sliders.

The interface also allows the user to select the crossover mechanism, the selection mechanism used in crossover, the size (at startup) of the population, the scope (global, local) of the crossover and the duration of the samples generated.

6 Other Interfaces

Over time it became clear that even with an X windows interface, with the faster generation that a note based system made possible, and with the more interesting samples generally produced by the note based system it was difficult to use the program for long periods of time and finding volunteers who were willing to do so was difficult. This was especially the case as the workstations available here are mostly in public areas where playing sound files is not considered particularly polite.

Another way for people to experiment with this was needed. One interface that could provide access to many people with only minimal interaction on each users part was the World Wide Web. The Web (and in particular a Mosaic client) provides access to sounds and support for users to vote on how much they liked the specific sounds presented. Each user was presented with six samples (from a total population of 36) and when all the samples had been rated, a new generation was built. The fact that the program read text input and wrote out sound files made the construction of a Web interface almost trivial. The interface was announced in various locations and a run of the program with 100 generations was achieved in just over a week. Since then, it has been far slower with only about 50 generations in about two months (though the fact that this was over the summer might have something to do with it).

Currently (end of August) a port of the system to Linux is underway and the complete source code should be available soon.

7 Results

Both the waveform and the note based programs remain difficult to use and the time needed to generate interesting music seems to be long enough to strain the patience of any user. However, in both cases some interesting samples were generated.

The primary problems with these programs seem to be those involving the human interface. The time it takes to run the program and listen to the samples combined with the relative dullness of the samples mean that the users rapidly get tired and bored. This means that the programs need to be built to generate the samples relatively quickly, the samples need to be relatively short, the samples need to develop and evolve to be interesting quickly and of necessity the population size must be small.

The Mosaic interface allows more people to use the program and thus usually allows for more input and more generations without the difficulties of a single user having to listen to many generations of (often quite uninteresting) noises. The results of 50 generations of the second Mosaic experiment are revealing (Figure 7). The figure shows the *most* complex of all the indi-

```
nextX ⇒ (cos 0.395849)
nextY ⇒ last-z
nextZ ⇒ last-freq
freq ⇒ (exp (cos (cos (minus (mod (sin last-y) (plus (invert last-delta)
  (negate 0.073791))) (xnl (times 0.427850 (max last-freq 0.009221)) (max (sin 0.198377)
  (exp last-amp)) (plus (exp last-amp) 0.266912) (xnl (normal last-amp last-amp)
  (or last-delta last-amp)) (negate last-freq) (xnl 0.524512 (times last-y last-delta)
  0.943715 0.283870 last-delta) (times (plus 0.228194 last-z) (plus last-y 0.385924)))
  0.666205)))))
amplitude ⇒ last-delta
duration ⇒ last-delta
deltaT ⇒ last-delta
```

Figure 3: Individual after 50 generations of Mosaic sound generation

viduals in the 50'th generations. The functions derived are mostly constants or very simple dependencies (for example `amplitude = last-delta`).

This would seem to indicate that constants generate the most interesting music, but the music generated generated also seems dull (at least to the author's ear). Why then does this happen? One reason is that when initial functions are small and the population is small, sometimes evolution leads to smaller functions over time and this trend can lead to functions that consist eventually of a single terminal. (Using deeper trees initially produces less interesting initial noises.) The other possibility is more subtle and more difficult to verify. As part of the process of constructing and verifying the genetic programming engine for this project, the author built a program that generated images. While running this program, it became clear that it wasnt sufficient to just rank images by how nice they were, it was also necessary to begin to understand the evolutionary mechanisms involved and to be aware of the effects of that process on the population as a while. Thus, it was sometimes productive to rate individuals highly not because they were attractive in some way, but because they seemed to offer potential for combining with other images, or sometimes just because they offered complexity that seemed to be otherwise missing.

The same effect seems to be present in the music program. The Mosaic program offers users no feedback, nor very much context. Thus, the users cannot choose to select for interesting samples because they seem promising for future crossovers and evolution.

In contrast to the Mosaic results, some of the functions from an individual picked at random from the 10th generation of selection by the author are presented in Figure 7. (In this figure the first few functions were chosen for display as the entire individual was quite large. These show much more complex functions with much more interaction. The sounds generated are substantially more interesting as well (but then they were selected to be interesting to the author). At times during the selection process individuals were rated highly not because they were interesting, but because they seemed to offer potential for future evolution or combinations.

8 Wrap Up

On the whole, this experiment cannot be said to be a success. Howerver, its not a complete failure either. The music generated does evolve and does seem to evolve toward sounds that are more interesting. Some of the samples even contain bits of what might be called melody though these are often ephemeral and get lost in the evolutionary process.

It seems worthwhile to continue the experiment. Several things need to be tried. One is to run the program often and long enough to find a number of good individuals to use as an initial population so that the first few generations are not as annoying and dull. (This is a good part of the motivation to distribute the program on PC's.)

Another related idea is to generate and save a number of interesting individuals and to examine them for common subexpressions - then to code up those subexpressions as primitives for further runs. This need for generating interesting individuals is part of the motivation for building versions of the program that can be distributed - initially for Linux PC's, in the long run for Windows PC's.

References

[1] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, pages 807–820, September 1988.

```
nextX ⇒ (normal (invert (minus last-freq last-duration))
  (min (min (plus last-freq (cos 0.986664)) (and (plus (invert 0.510795)
  (exp (xor (plus (minus last-amp (exp (negate (times last-x last-x)))) last-duration)
  (plus last-y (times last-y last-delta))))) (times last-x 0.289039)))
  (invert (min last-freq last-z))))
nextY ⇒ (sin 0.926196)
nextZ ⇒ (minus (min last-y 0.605259) (minus (min (negate last-freq)
  (invert (min last-y 0.605259))) (min (min (negate (not (sin (cos (max 0.374724
  (minus 0.437766 last-x)))))) (cos (minus last-duration last-z))) (not (exp last-freq)))))
freq ⇒ (if (sin last-delta) (xnl (not (mod last-z (exp last-delta))) 0.320894
  (max (mod last-x (invert (invert (xor (min last-z 0.158901)
  last-delta)))) (minus 0.863744 0.578743)) 0.685628 (minus (plus last-x
  (negate (sin last-delta))) (xnl (not (mod last-z (exp last-delta)))
  0.320894 (max (mod last-x (invert (invert (xor (min last-z 0.158901)
  last-delta)))) (xor (cos (if (sin last-z) (minus (xnl 0.727900 (xnl
  last-z 0.792953 (negate last-x) 0.678791 last-freq) last-z (max
  last-freq 0.699761) last-delta) (plus (plus (max 0.842791
  last-duration) last-amp) (max last-amp (min last-duration last-amp))))
  (xor (max last-amp (invert (times last-x last-x))) (sin last-x))))
  (max (xnl (sin last-freq) (minus 0.701262 last-delta) (min (xnl
  last-freq last-delta 0.370010 last-duration last-z) (exp
  last-duration)) (minus (min (invert (sin (and (invert last-x) (xnl
  last-freq last-y 0.074343 last-delta last-z)))) (exp (plus (minus
  0.453761 0.317075) (plus last-amp 0.145036)))) last-freq) (plus (min
  last-amp 0.250144) last-y)) (max (cos last-y) last-delta)))) 0.685628
  (minus (plus last-x 0.022382) (max (rand last-delta) (or (plus (mod
  0.739967 last-y) (cos 0.167086)) (exp (cos last-delta))))))) (plus
  (min (sin (exp last-x)) (exp last-duration)) last-duration))
amplitude ⇒ (max 0.959349 (times 0.959349 (max last-amp (times (or
  (divide (times last-z last-y) last-delta) last-z) (sin (exp last-delta))))))
```

Figure 4: Typical Functions from an Individual generated by author after 10 generations

[2] John Koza. *Genetic Programming.* MIT Press, 1992.

[3] John Osterhout. *Tcl and the Tk Toolkit.* Addison Wesley, 1994.

[4] Karl Sims. Artificial evolution for computer graphics. *Computer Graphics*, 25(4):319–328, 1991.